Modélisation compositionnelle d'architectures GALS dans un modèle de calcul polychrone

Yue Ma, Thierry Gautier, Jean-Pierre Talpin, Paul Le Guernic, Huafeng Yu

Modélisation compositionnelle d'architectures GALS dans un modèle de calcul polychrone

Yue Ma — Thierry Gautier — Jean-Pierre Talpin — Paul Le Guernic — Huafeng Yu

INRIA Rennes/IRISA
Campus de Beaulieu, 35042 Rennes, France
{firstname}.{lastname}@inria.fr

RÉSUMÉ. *AADL est dédié à la conception de haut niveau et l'évaluation de systèmes embarqués. Il permet de décrire la structure d'un système et ses aspects fonctionnels par une approche à base de composants. Des processus localement synchrones sont alloués sur une architecture distribuée et communiquent de manière globalement asynchrone (système GALS). Une spécificité du modèle polychrone est qu'il permet de spécifier un système dont les composants peuvent avoir leur propre horloge d'activation : il est bien adapté à une méthodologie de conception GALS. Dans ce cadre, l'atelier Polychrony fournit des modèles et des méthodes pour la modélisation, la transformation et la validation de systèmes embarqués. Cet article propose une méthodologie pour la modélisation et la validation de systèmes embarqués spécifiés en AADL via le langage synchrone multi-horloge Signal. Cette méthodologie comprend la modélisation de niveau système en AADL, des transformations automatiques du modèle AADL vers le modèle polychrone, la distribution de code, la vérification formelle et la simulation du modèle polychrone. Notre transformation prend en compte l'architecture du système, décrite dans un cadre IMA, et les aspects fonctionnels, les composants logiciels pouvant être mis en œuvre en Signal. La génération de code distribué est obtenue avec Polychrony. Une étude de cas illustre notre méthodologie pour la conception fiable des applications AADL.*

ABSTRACT. *AADL is dedicated to high-level design and evaluation of embedded systems. It allows describing both system structure and functional aspects via a component-based approach, e.g., GALS system. The polychronous model of computation stands out from other synchronous specification models by the fact that it allows one specifying a system whose components can have their own activation clocks. It is well adapted to support a GALS design methodology. Its framework Polychrony provides models and methods for modeling, transformation and validation of embedded systems. This paper proposes a methodology for modeling and validation of embedded systems specified in AADL via the multi-clock synchronous programming language*

JESA – 45/2011. MSR 2011, pages 61 à 76

Signal. This methodology includes system-level modeling via AADL, automatic transformations from the high-level AADL model to the polychronous model, code distribution, formal verification and simulation of the obtained polychronous model. Our transformation takes into account both the system architecture, particularly described in Integrated Modular Avionics (IMA), and functional aspects, e.g., software components implemented in the polychronous language Signal. AADL components are modeled into the polychronous MoC within the IMA architecture using a library of ARINC services. Distributed code generation is obtained with Polychrony. A case study illustrates our methodology for the reliable design of AADL applications.

MOTS-CLÉS : *AADL, Signal, GALS, IMA, modèle polychrone, simulation.*

KEYWORDS: *AADL, Signal, GALS, IMA, polychronous model, simulation.*

1. Introduction

Les systèmes embarqués se rencontrent aujourd'hui de façon universelle, y compris dans la vie quotidienne, et sont notamment utilisés pour effectuer des tâches critiques (Knight, 2002) dans des domaines tels que l'avionique, l'automobile et les télécommunications. Un système embarqué temps réel est un système dont les actions sont assujetties à des délais temporels. Un défaut de respect des délais peut compromettre l'exactitude des résultats et avoir des conséquences graves (temps réel strict) (Laplante, 1992).

Le langage AADL (Architecture Analysis and Design Language) est proposé comme support à une nouvelle méthodologie pour la conception de systèmes embarqués. En AADL, la partie matérielle, comprenant mémoires, processeurs, bus, etc., peut être modélisée sous forme logicielle, ce qui offre une plus grande souplesse car cela permet des prototypages et expérimentations sans nécessairement disposer d'une implantation physique du système. En outre, les approches à base de composants, comme c'est le cas en AADL, fournissent un moyen de réduire considérablement les coûts de développement via la modularité et la réutilisation de composants.

Les approches les plus récentes tendent à introduire dans les modèles un nombre croissant d'éléments asynchrones, tout en essayant de préserver les bonnes propriétés des modèles synchrones. On obtient ainsi un mélange de modèles de conception synchrones et asynchrones, comme par exemple le modèle GALS (Globalement Asynchrone Localement Synchrone (Daniel, 1984)). Rassemblant les avantages du synchrone et de l'asynchrone, le modèle GALS est en train de devenir une architecture de choix pour la mise en œuvre de spécifications complexes à la fois matérielles et logicielles. Le problème de la validation de l'ensemble du système est essentiel : l'exécution du logiciel sur l'architecture cible est généralement asynchrone, mais cette phase de la conception est la plus sujette aux erreurs.

Dans cet article, nous proposons une méthodologie de modélisation de la composition globalement asynchrone de composants synchrones. Nous nous plaçons dans

le cadre de la programmation synchrone multi-horloge, ou *polychrone* (Le Guernic *et al.*, 2002), en considérant plus particulièrement les architectures de type IMA (ARINC651, 1997). Un des principaux objectifs de l'approche est d'étudier les propriétés des systèmes globalement asynchrones en utilisant des outils de simulation et de vérification de modèle (*model-checking*) existant dans le cadre synchrone.

Notre solution peut être considérée comme une transformation d'une conception en composants localement synchronisés connectés de façon asynchrone, vers un modèle synchrone. On utilise en premier lieu le langage AADL pour modéliser une application temps réel. Étant donné la sémantique des composants AADL dans le modèle de calcul polychrone, le modèle AADL est alors transformé en modèle Signal . À partir de ce modèle, l'outil Polychrony peut être utilisé pour effectuer de la vérification, des simulations et d'autres analyses. Sachant qu'il peut être non trivial de représenter de façon adéquate l'asynchrone et le non déterminisme dans un cadre synchrone, nous proposons une méthode qui utilise des techniques et des bibliothèques existant dans l'environnement de Signal, consistant notamment en un modèle des services du système d'exploitation temps réel APEX-ARINC-653 (ARINC653, 1997).

2. Background : AADL, architectures avioniques et Polychrony

2.1. *Abstractions du langage AADL*

Le langage AADL est une norme de la SAE, qui a été défini pour les systèmes embarqués critiques temps réel. Il doit rendre possible l'utilisation d'approches formelles diverses pour l'analyse de systèmes constitués de composants matériels et logiciels.

Catégories de composants : Pour modéliser les systèmes embarqués complexes, AADL fournit trois catégories de composants : 1) Les données, les sous-programmes, les « *threads* »[1] et les *processus AADL* représentent collectivement le logiciel d'une application ; ils sont appelés composants logiciels. 2) Les composants de la plate-forme d'exécution (processeurs, dispositifs (*device*), etc.) supportent l'exécution des threads, le stockage des données et du code, et la communication entre les threads. 3) Les systèmes sont appelés composants composites. Ils permettent d'organiser en structures hiérarchiques avec des interfaces bien définies les composants logiciels et les composants de la plate-forme d'exécution.

Un exemple permet de donner une première idée de AADL. Le système SDSCS (Simplified Doors and Slides Control System) (Yu *et al.*, 2011) est une version générique simplifiée du système qui permet de gérer les portes des avions de la série Airbus A350. Ce système se concentre sur la gestion de deux portes passagers. Chaque porte passager dispose d'un composant logiciel qui réalise un ensemble de tâches. Le système est présenté comme un système AADL. Les deux portes sont modélisées comme des sous-systèmes. Ils sont contrôlés par deux *processus AADL*. Chaque processus *doors_process* (figure 1) se compose de trois threads : *door_handler1*, *door_handler2*

1. Nous gardons ici le terme anglais plutôt que « fil d'exécution »ou « tâche ».

et *doors_mix*, qui calculent et génèrent les sorties. La modélisation détaillée du système est illustrée dans (Ma, 2010). Dans cette étude de cas, AADL est principalement utilisé pour la modélisation de l'architecture du système. La modélisation fonctionnelle devrait être ouverte à d'autres languages de modélisation de haut niveau.

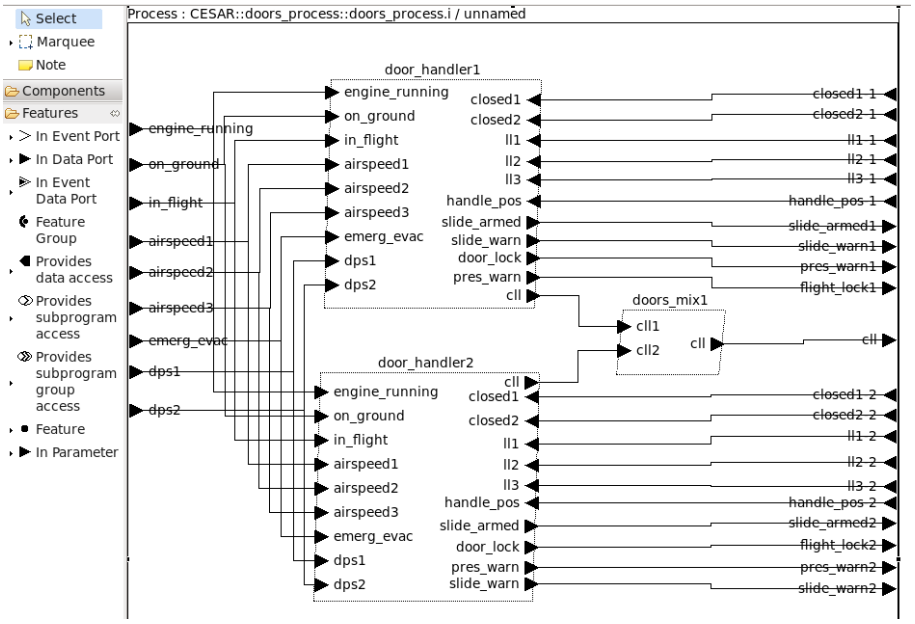

Figure 1. *Les threads du* processus AADL doors_process

Domaine temporel : Trois événements (t_d, t_s, t_f) sont associés à chaque thread t. L'événement t_d (*« dispatch »*) est l'horloge à laquelle le thread est « réveillé » ; t_s (*« start »*) (resp., t_f (*« deadline »*)) est l'horloge à laquelle le thread démarre (resp., doit se terminer). Ces deux événements, t_s et t_f, peuvent être spécifiés respectivement par les propriétés *Input_Time* et *Output_Time*. De nombreuses autres caractéristiques temporelles, par exemple l'échéance, le temps d'exécution, etc., peuvent être spécifiées par les propriétés définies dans la norme AS5506 (SAE AS5506, 2004) (SAE AS5506A, 2009).

2.2. *Architecture avionique et ARINC 653*

IMA (Integrated Modular Avionics) est une approche nouvelle pour les architectures des systèmes avioniques, qui introduit des méthodes pouvant atteindre de hauts niveaux de réutilisation avec un coût limité. Une caractéristique forte des architectures IMA est le fait que plusieurs applications avioniques peuvent être hébergées sur un seul système, partagé. Ces applications sont assurées qu'aucune propagation de faute ne se produise d'un composant à un autre. Ce problème est résolu au moyen d'un

partitionnement fonctionnel des applications selon la disponibilité des ressources en temps et en mémoire.

ARINC 653 est un standard définissant notamment des interfaces de programmation pour les logiciels applicatifs avioniques suivant une architecture IMA. Une *partition* est composée de *processus ARINC* qui représentent les unités d'exécution. Le *système d'exploitation de niveau partition* est responsable de l'exécution correcte des *processus ARINC* dans une *partition*. Des mécanismes appropriés sont prévus pour la communication et la synchronisation entre *processus ARINC* (*buffers, événements, sémaphores*) et entre *partitions* (*ports* et *canaux*). La norme avionique ARINC définit les principes de base du partitionnement, ainsi qu'un ensemble de services, conformes à l'architecture IMA.

AADL peut être utilisé en particulier pour une modélisation conforme à ARINC. AADL et ARINC ont certaines caractéristiques similaires. La *partition* ARINC est proche du *processus AADL*. Le *processus AADL* représente un espace d'adressage protégé, un espace de partitionnement où une protection est assurée contre les accès d'autres composants à l'intérieur du processus. Ainsi, la *partition* ARINC et le *processus AADL* sont des unités du partitionnement. Les mécanismes de communication définis dans ARINC supportent les communications de messages en file et sans file.

2.3. *Langage Signal et modélisation ARINC en Signal*

L'approche synchrone est l'une des solutions possibles pour une conception sûre des systèmes embarqués. Le modèle multi-horloge ou polychrone se distingue des autres modèles synchrones par son cadre uniforme. Il permet aussi bien la conception de systèmes dans lesquels chaque composant dispose de sa propre horloge d'activation, que de systèmes mono-horloge. Cette caractéristique rend la sémantique de Signal (Benveniste *et al.*, 1991) plus proche de la sémantique de AADL que ne l'est celle d'autres modèles purement synchrones ou asynchrones. Cela facilitera ainsi la validation du système.

2.3.1. *Le langage Signal*

Signal est un langage relationnel qui s'appuie sur le modèle polychrone. Signal manipule des suites non bornées de valeurs typées $(x_t)_{t\in\mathbb{N}}$, appelées *signaux*, notées $\mathtt{x}$ et indexées implicitement par les valeurs discrètes de leur horloge, notée $\hat{x}$. À un instant donné, un signal peut être présent ou absent. Deux signaux sont dits synchrones s'ils sont toujours présents (ou absents) aux mêmes instants.

En Signal, un processus Signal (dénoté P ou Q) consiste en la composition synchrone (notée $P|Q$) d'équations sur signaux (notées $x := y\, f\, z$). Une équation $x := y\, f\, z$ définit le signal de sortie x par la relation sur ses signaux d'entrée y et z auxquels est appliqué l'opérateur f. Le processus P/x restreint la portée du signal x au processus P. La syntaxe abstraite d'un processus Signal P en Signal est définie comme suit : $P, Q \ ::= \ x := y\, f\, z \ \mid \ P|Q \ \mid \ P/x$

Nous discutons ici certains écarts et similitudes sémantiques entre AADL et Signal. Signal peut fournir un outillage à base d'opérateurs dérivés permettant de réduire les écarts.

– **Outillage d'horloges**. Les horloges peuvent être étroitement liées aux domaines temporels AADL et des opérateurs sont définis en Signal pour manipuler les horloges.

– **Outillage pour le report des communications**. Le report des communications peut être implémenté en Signal à l'aide de cellules mémoire et de FIFO.

– **Outillage pour briser l'atomicité**. Pour un programme Signal, l'horloge la plus rapide d'un processus Signal n'est pas toujours une horloge d'entrée. Le sur-échantillonnage permet la spécification de contraintes entre les entrées et les sorties de telle sorte qu'il ne puisse pas se produire de valeur d'entrée supplémentaire tant que les contraintes en question ne sont pas respectées par les calculs de la sortie (Le Guernic *et al.*, 2002).

2.3.2. *Modélisation de concepts ARINC en Signal*

Les applications avioniques qui s'appuient sur la norme avionique ARINC 653, basée sur l'architecture IMA, peuvent être spécifiées dans le modèle de Signal. Une bibliothèque de services APEX ARINC est fournie en Signal. L'environnement de conception Polychrony comprend ainsi une bibliothèque Signal de composants correspondant aux services d'exécutif temps réel définis dans ARINC (ARINC653, 1997). Cette bibliothèque, conçue par Abdoulaye Gamatié (Gamatié, 2004), s'appuie sur quelques blocs de base (Gamatié *et al.*, 2003), qui permettent de modéliser les *partitions* : il s'agit des services APEX-ARINC-653, d'un modèle de RTOS, et d'entités d'exécution.

3. Modélisation de composants AADL en processus Signal

Cette section se concentre sur la modélisation synchrone des composants AADL dans une architecture IMA, de sorte qu'un modèle AADL puisse être traduit en un modèle exécutable du langage flot de données polychrone Signal. Cette modélisation est une contribution qui doit aider à combler le fossé existant entre modèles asynchrones et synchrones.

3.1. *Chaîne et Principes de transformation*

Nous décrivons la transformation de AADL en isolant les catégories syntaxiques de base qui caractérisent ses capacités expressives : systèmes, *processus AADL*, threads, sous-programmes, données, dispositifs, processeurs, bus et connexions. Le modèle Ecore de AADL est traduit en un modèle SME (ESPRESSO, INRIA, 2011) (SME est un métamodèle du langage Signal) par définition de règles de transformation. Le modèle SME peut ensuite être transformé en programme Signal. Le pro-

gramme Signal est alors compilé et un code exécutable C (ou Java/C++) peut être généré.

AADL	**Signal**
thread	*processus ARINC*
processus AADL	*partition* ARINC
port	FIFO bornée
connexion de ports de données	processus Signal (selon le type de la connexion)
processeur	*OS de niveau partition* en ARINC
type de données	type de données Signal
système	processus Signal constitué de sous-processus
bus	processus Signal de communication
dispositif	processus Signal

Tableau 1. *Principes de base de la transformation*

Cette transformation de modèle est basée sur l'étude des similarités entre AADL et les services APEX-ARINC. La transformation d'un modèle AADL vers Signal repose sur l'architecture IMA (Ma *et al.*, 2008). Les principes de base de la transformation de base sont présentés dans la table 1.

3.2. *Du temps logique abstrait vers un temps de simulation concret*

Le paradigme synchrone fournit une représentation idéalisée du parallélisme. Tandis que AADL prend en compte les durées de calcul et les délais de communication, permettant ainsi de produire des données d'un même instant logique à des instants différents dans la mise en œuvre. Pour faire le lien entre les deux modèles, nous conservons la vision idéale de calculs et communications instantanés, et déportons le calcul des latences et des délais de communication vers des processus Signal spécifiques de « mémorisation », qui introduisent les retards et synchronisations requis.

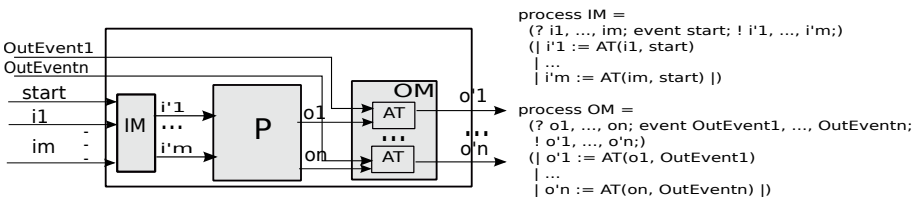

Figure 2. *Modélisation d'une tâche consommant du temps*

Une caractéristique principale des programmes polychrones est l'exécution logique instantanée, par rapport au temps logique. Alors qu'en AADL, un thread peut exécuter une fonction ou un calcul pendant un intervalle de temps spécifié, défini par

les propriétés temporelles. Par conséquent, la modélisation de AADL dans le cadre polychrone nécessite une forme d'adaptateur pour interfacer le temps logique abstrait et le temps de simulation concret. À chaque sortie o d'un processus Signal P, nous associons un processus Signal OM dont la sortie est la valeur de o retardée jusqu'à ce que son *Output_Time* représenté par le signal événement d'entrée *OutEvent* se produise (figure 2, le processus Signal AT() est défini comme une mémoire, la définition détaillée peut être trouvée dans (Ma, 2010)). Ainsi, à chaque processus Signal P, nous associons un signal événement d'entrée « *start* »pour modéliser les délais de propagation, et l'exécution de P est synchronisée avec *Start* (par le processus Signal IM).

3.3. *Modélisation de thread*

Les threads sont les principaux composants AADL exécutables et ordonnançables. Un thread $Th = < P,\ C,\ R,\ Su,\ T/S >$ représente une unité concurrente ordonnançable d'exécution séquentielle définie par un code source. Un thread Th encapsule une fonctionnalité qui peut consister en des ports P, des connexions $C = P \times P$, des propriétés R, des spécifications de comportement T/S et des sous-programmes Su qui peuvent être invoqués par le thread.

La sémantique d'exécution d'un thread AADL est la suivante : 1) Lire et geler les entrées. Le contenu des données entrantes est gelé pour la durée d'exécution du thread. L'entrée est gelée au moment spécifié par *Input_Time* (dispatch par défaut). Toute entrée arrivant après ce gel devient disponible à l'instant d'entrée suivant. 2) Exécuter et calculer. Quand l'activité du thread entre dans l'état de calcul, l'exécution de la séquence de code source est gérée par un ordonnanceur. 3) Actualiser et rendre disponibles les sorties. Par défaut, une sortie est transférée vers d'autres composants à l'instant de terminaison, ou comme spécifié par la valeur de la propriété *Output_Time*.

3.3.1. *Étapes d'interprétation*

Un thread AADL Th est d'abord traduit en un processus Signal SP, qui correspond à un *processus ARINC*, du point de vue de la structure fonctionnelle. SP a les mêmes flots d'entrée/sortie que Th. En raison de la différence de sémantique temporelle entre AADL et Signal, les propriétés temporelles d'un thread AADL sont traduites par un autre processus Signal, SPT (figure 3), qui joue le rôle d'interface sémantique temporelle entre AADL et Signal. Les principales fonctions de SPT en regard de SP sont d'activer le processus Signal fonctionnel SP au moment du *start*, et mémoriser les sorties du thread jusqu'à la date de terminaison.

Le domaine temporel associé à l'exécution d'un thread (Section 2.1) est défini par certaines caractéristiques temps réel : (t_d, t_s, t_f). En conformité avec la sémantique du contrôle temps réel, un nouveau processus Signal, SPS (figure 3), est ajouté, à l'intérieur duquel les signaux de contrôle temporel sont automatiquement calculés lorsqu'il est activé. Quand il reçoit les signaux gérant l'ordonnancement (par exemple, *dispatch*, qui correspond à horloge t_d) depuis l'ordonnanceur de threads, il commence à

calculer ses propres signaux temporels pour l'activation et la complétion du processus Signal SPT.

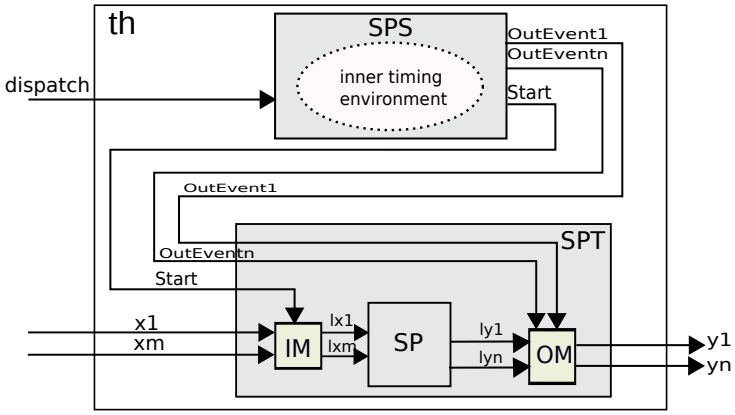

Figure 3. *Traduction d'un thread AADL en un processus Signal*

3.3.2. *Règles d'abstraction*

Nous donnons une description abstraite des règles de transformation d'un thread $Th = < P,\ C,\ R,\ Su,\ T/S >$ en un processus Signal (en tant que *processus ARINC*). La notation $\mathcal{I}(X)$ représente l'interprétation d'un composant X.

1) Un composant thread est traduit par un *processus ARINC*, paramétré par $\mathcal{I}(P),\ \mathcal{I}(C),\ \mathcal{I}(R),\ \mathcal{I}(Su),\ \mathcal{I}(T/S)$. La notation $Program^{parameters}$ est utilisée pour dénoter que les paramètres *parameters* serviront à la définition du *Program*. La fonctionnalité du thread est traduite en deux sous-processus, SPS et SPT, à l'intérieur du processus Signal.

$\mathcal{I}(Th) = (SPS^{\mathcal{I}(R)} \mid SPT^{\mathcal{I}(P),\ \mathcal{I}(C),\ \mathcal{I}(Su),\ \mathcal{I}(T/S)})$

2) Le sous-processus SPS est paramétré par les propriétés R. L'interface de SPS inclut les entrées *dispatch* et les sorties *start* (correspondant à l'horloge t_s), *complete* (l'horloge à laquelle le thread se termine), *deadline* (correspondant à l'horloge t_f). Les propriétés $r \in R$, par exemple, *Input_Time*, *Output_Time*, sont interprétées dans SPS. L'interface du sous-processus SPS est décrite comme :

$process\ SPS = (?\ event$ dispatch; $!\ event$ start, complete, deadline; $\ldots;)$.

3) Le sous-processus $SPT^{\mathcal{I}(P),\ \mathcal{I}(C),\ \mathcal{I}(Su),\ \mathcal{I}(T/S)}$ est paramétré par $\mathcal{I}(P),\ \mathcal{I}(C),\ \mathcal{I}(Su),\ \mathcal{I}(T/S)$. Les connexions, sous-programmes et comportements sont traduits dans le sous-processus SPT.

4) Les ports d'entrée/sortie P du thread sont traduits en entrées/sorties de Signal. L'interface du *processus ARINC* inclut ces signaux et les entrées événements *dispatch* reçues de l'ordonnanceur. La valeur p.*direction* désigne la direction d'un port p.

$inputs = \{\text{dispatch}\} \cup \mathcal{I}(p) \quad \forall p \in P,\ p.direction = \mathbf{in}$

$outputs \ = \ \{\mathcal{I}(p)\} \quad \forall p \in P, \ p.\textit{direction} = \mathbf{out}$

5) Chaque sous-programme $su \in Su$ est interprété comme un processus Signal PP_{su} :

$\mathcal{I}(su) \ = \ process \ PP_{su}(?\ In1, \ldots Inm; !\ Out1, \ldots Outn;)\ PP_{su}\_BODY.$

Il est invoqué comme une instance dans le sous-processus SP :

$(|\ o1 := Out1 \mid \ \ldots \mid on := Outn \mid PP_{su}| \ In1 := i1 \mid \ \ldots \mid Inm := im)$

6) Une connexion dans un thread $c = (p_1, p_2) \in C$ relie les ports (paramètres) des processus Signal correspondant aux sous-programmes invoqués. Une telle connexion est interprétée comme une affectation reliant les entrées/sorties des modèles de sous-programmes :

$\mathcal{I}(p_2) := \mathcal{I}(p_1)$

7) Les transitions/actions T/S sont interprétées dans un processus Signal synchrone SP. Ce processus synchrone SP est encapsulé dans le sous-processus SPT. La valeur $c.type$ désigne le *type* de la connexion c, qui peut être *immediate* ou *delayed*.

$SPT \ = \ (IM \mid SP^{\mathcal{I}(Su),\mathcal{I}(C)} \mid OM)$

où : $SP^{\mathcal{I}(Su),\mathcal{I}(C)} \ = \ \mathcal{I}(T/S)$

$IM \ = \ (ii_{p_{k_1}} := AT(\mathcal{I}(p_{k_1}),\ \text{start}) \mid ii_{p_{k_2}} := AT(\mathcal{I}(p_{k_2}),\ \text{start}) \mid ...)$

$\forall p_{k_j} \in P,\ t.q.\ p_{k_j}.\textit{direction} = \mathbf{in}$

$OM \ = \ (OM_{imm} \mid OM_{delayed})$

$OM_{imm} = (oo_{p_{k_1}} := AT(\mathcal{I}(p_{k_1}),\ \text{complete}) \mid oo_{p_{k_2}} := AT(\mathcal{I}(p_{k_2}), \text{complete}) \mid ...)$

$\forall p_{k_j} \in P,\ t.q. \exists c \in C,\ c = (p_1, p_{k_j}), c.\textit{type} = \mathbf{immediate}, p_{k_j}.\textit{direction} = \mathbf{out}$

$OM_{delayed} = (oo_{p_{k_1}} := AT(\mathcal{I}(p_{k_1}), \text{deadline}) \mid oo_{p_{k_2}} := AT(\mathcal{I}(p_{k_2}), \text{deadline}) \mid ...)$

$\forall p_{k_j} \in P,\ t.q.\ \exists c \in C,\ c = (p_1, p_{k_j}),\ c.\textit{type} = \mathbf{delayed},\ p_{k_j}.\textit{direction} = \mathbf{out}$

Nous ne présentons ici que la modélisation du thread. Le principe de modélisation des autres composants est similaire, et les détails peuvent être obtenus dans (Ma, 2010).

4. Génération de modèles de simulation distribués

Un système distribué peut être beaucoup plus étendu et puissant que des combinaisons de systèmes autonomes (Anderson, 2001). Polychrony fournit des méthodes générales de compilation, en particulier pour la génération de code distribué (Aubry *et al.*, 1996).

Pour une transformation complète, nous considérons deux spécifications. La première est le programme Signal qui comprend tous les composants de calcul et représente les flots de données, programme traduit à partir des composants AADL. La

seconde est la spécification de l'architecture matérielle : dans la spécification d'architecture AADL, la façon dont le système doit être distribué est clairement définie.

Pour distribuer semi-automatiquement un programme Signal, tout d'abord, les composants spécifiés sont placés (*mapping*) sur l'architecture cible, un ordonnanceur est ajouté pour chaque processeur, et les synchronisations d'horloges entre les composants partitionnés sont synthétisées. Le mécanisme des pragmas en Signal (pragma « RunOn »(Besnard *et al.*, 2011)) est utilisé pour représenter les informations de placement. Ensuite, des modèles de communication sont ajoutés entre ces composants. L'étape finale consiste en la génération des codes exécutables distribués (Ma *et al.*, 2009).

4.1. *Placement*

Comme un *processus AADL* est traduit en une *partition* ARINC, et qu'il est lié à un processeur comme spécifié dans les *propriétés*, on peut affecter chaque *partition* à un processeur donné. Chaque *partition* est rythmée par sa propre horloge. Le pragma Signal « RunOn »est utilisé dans l'environnement Polychrony pour les informations de partitionnement : quand un partitionnement basé sur l'utilisation du pragma « RunOn »est appliqué à un système, l'application globale est partitionnée selon les différentes valeurs du pragma de façon à obtenir des sous-modèles de processus Signal. L'arbre des horloges (la racine de l'arbre représente l'horloge la plus fréquente) et l'interface de ces sous-modèles peuvent être complétés de manière à ce qu'ils représentent des processus Signal *endochrones* (Talpin *et al.*, 2008) (un processus endochrone est insensible aux délais de propagation internes et externes).

Les principes du placement sont les suivants :

1) Description du flot de données du logiciel, en utilisant le langage Signal (programme P).

2) Description de l'architecture cible. Celle-ci est spécifiée dans le modèle AADL, en particulier dans les *propriétés*.

3) Placement du programme logiciel (correspondant à la partie des composants logiciels AADL) sur l'architecture cible. Le programme P est ainsi récrit sous la forme `(|P1|...|Pn|)`, où n est le nombre de processeurs. Cette étape n'est qu'une restructuration syntaxique du programme d'origine. Chaque processus Signal Pi est annoté avec un pragma « RunOn i », utilisé pour le partitionnement.

4.2. *Ordonnanceur*

L'ordonnanceur sélectionne les tâches à exécuter en fonction de la politique d'ordonnancement. Dans le modèle APEX-ARINC, l'*OS de niveau module* est l'ordonnanceur qui va ordonnancer les *partitions* à l'intérieur d'un même module. L'ordonnancement des *partitions* est strictement déterministe (du point de vue du temps) dans

la norme ARINC 653. Chaque *partition* est ordonnancée selon sa fenêtre temporelle de partition. À l'intérieur de la *partition*, un *OS de niveau partition* est utilisé pour l'ordonnancement des *processus AADL* .

Nos règles de transformation suivent la norme AADL, selon laquelle chaque processeur peut être attribué à plusieurs *processus AADL* , ce qui signifie qu'un processeur peut exécuter plusieurs *partitions*. Comme l'ordonnanceur des *partitions* s'exécutant sur un processeur est déterministe, il est statique et fixe selon les tables de configuration. Nous créons à cette fin un ordonnanceur statique simple (cf. *Module_Level_OS* de APEX-ARINC). Le compilateur utilise le *calcul d'horloges* pour analyser statiquement chaque équation du programme, pour identifier la structure des horloges des variables, et pour ordonnancer l'ensemble des calculs. Si la composition des équations contient des contraintes d'horloges, cela signifie qu'il existe des contraintes de synchronisation. Le programme distribué est généré automatiquement dans l'environnement Polychrony, en appliquant la compilation en mode *distribution*. Par rapport au programme d'origine, une réécriture a été effectuée pour les parties séparées. Toutes les propriétés de calcul sont préservées, et un ordonnanceur a été ajouté pour chacun des processeurs.

4.3. *Ajout des communications*

Un programme distribué consiste en un ensemble de processus Signal interagissant avec l'environnement et communiquant entre eux. Dans notre implémentation, nous utilisons MPI (Message Passing Interface), qui est un protocole du communication indépendant du langage. Il s'agit d'une interface de passage de messages, ainsi que de protocoles et de spécifications sémantiques pour la façon dont ses fonctions doivent se comporter dans une mise en œuvre. Les messages permettant de communiquer dans les systèmes distribués peuvent être de type synchrone ou asynchrone. Pour exécuter les programmes distribuées, il est nécessaire de configurer les machines physiques pour leur exécution. Finalement, les parties peuvent être compilées localement et exécutées respectivement sur les différentes machines.

5. Simulation

Dans cette section, nous décrivons le processus de conception pour la co-modélisation et la co-simulation d'une étude de cas du projet CESAR (Yu *et al.*, 2011) dans le cadre de Polychrony. Ce processus de conception est basé sur une approche de co-conception au niveau système.

L'étude de cas SDSCS est un système simplifié de contrôle des portes et toboggans en avionique (voir Section 2.1). En plus de la simulation AADL, cette étude de cas illustre également une approche de co-conception de système de haut niveau et hétérogène, basée sur les modèles, pour une architecture de système globalement asynchrone et localement synchrone (GALS).

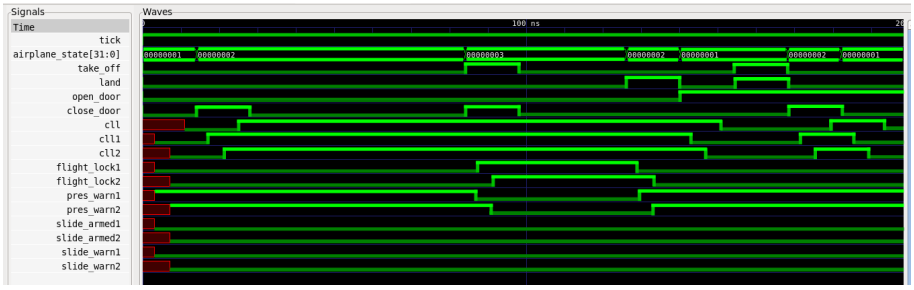

Figure 4. *Simulation avec afficheur VCD : GTKWave*

Dans cette étude de cas, l'architecture matérielle distribuée (ici, les deux *processus AADL* sont exécutés sur des processeurs différents), ainsi que les aspects asynchrones, sont modélisés en AADL, tandis que le comportement fonctionnel est modélisé en Simulink. SME/Polychrony est adopté comme formalisme commun permettant de faire la liaison entre deux modèles hétérogènes. Une démonstration de simulation de cette étude de cas par des afficheurs VCD (figure 4) a été réalisée dans le cadre de Polychrony. Les fichiers VCD sont utilisés pour la visualisation des résultats de simulation par les afficheurs VCD, tels que GTKWave. Sur la figure, le changement des valeurs des signaux par rapport à l'horloge la plus rapide apparaît. Une explication plus détaillée est donnée dans (Ma, 2010).

L'architecture de distribution décrite en AADL est préservée dans notre transformation. Chaque processus Signal (correspondant à un *processus AADL*) est configuré pour un processeur physique suivant la spécification AADL. Le code distribué est généré dans Polychrony pour simuler l'application sur l'architecture considérée.

Des vérifications formelles pouvant également être effectuées via Polychrony. Une étude de cas de système de guidage en vol est illustrée dans (Ma, 2010). Les modèles sont traduits dans une représentation symbolique, en l'occurrence, des systèmes dynamiques polynomiaux, manipulés par le *model checker* Sigali (Marchand *et al.*, 2000). Des propriétés de sûreté exprimées en CTL sont vérifiées sur le système avec Sigali.

6. Travaux connexes : formalisations de AADL

Le langage AADL fournit un bon support pour la description et l'analyse de systèmes embarqués complexes. Afin de valider des propriétés formelles sur un modèle AADL, d'effectuer des analyses d'ordonnançabilité ou de performance, d'effectuer de la vérification, un cadre formel qui puisse fournir des diagnostics sur le système doit être défini et utilisé.

– L'utilisation de MARTE pour modéliser AADL est un essai de description formelle des aspects AADL du temps en utilisant MARTE, et de définition d'un modèle

UML exécutable avec la sémantique d'exécution AADL, mais l'objectif d'un langage analysable formellement reste une perspective (Lee *et al.*, 2008).

– La traduction de AADL vers BIP permet la simulation de modèles AADL, ainsi que l'application de techniques de vérification. L'absence de localité dans BIP rend difficile le raisonnement compositionnel (Pi *et al.*, 2009).

– L'objectif de la modélisation ARINC 653 en AADL (Delange *et al.*, 2009) est de proposer un processus de développement de type MDE utilisant AADL pour la modélisation, la vérification et l'implémentation de systèmes basés sur ARINC 653. Nous avons une démarche inverse consistant à modéliser un système AADL en Signal dans un cadre ARINC 653.

– L'objectif principal de AADL2SYNC (Jahier *et al.*, 2007) est d'effectuer une simulation et une validation qui prennent en compte à la fois l'architecture du système et les aspects fonctionnels. Ce travail construit un simulateur exprimé dans un langage purement synchrone, Lustre, néanmoins sa capacité d'expression reste limitée.

Par rapport à ces travaux, notre approche a des objectifs multiples. Nous modélisons un système AADL en Signal afin d'effectuer de la simulation, et de la génération de code C, puisque le modèle polychrone du langage Signal offre en effet un support formel pour l'analyse, la vérification, la simulation, mises en œuvre dans Polychrony. En outre, le modèle polychrone fournit des modèles et des méthodes pour l'intégration rapide, basée sur le raffinement, et la vérification formelle de conformité des architectures GALS (Talpin *et al.*, 2003).

7. Conclusion

Nous proposons une méthodologie pour la modélisation de systèmes embarqués spécifiés en AADL via le modèle de calcul polychrone. Notre conception prend en compte à la fois l'architecture du système, plus spécialement pour une architecture décrite selon le modèle IMA, et les aspects fonctionnels, par exemple, avec des composants logiciels implémentés dans le langage de programmation synchrone Signal. À partir de spécifications AADL de haut niveau, la distribution automatique de code pour la simulation a été expérimentée via le modèle polychrone et les pragmas de distribution de Signal. Des simulations de haut niveau sont ensuite effectuées au moyen de technologies et outils associés, comme les démonstrateurs de changements de valeurs, qui permettent d'analyser le modèle résultant. Des techniques et des bibliothèques existantes de Polychrony, qui consistent en un modèle des services temps réel de APEX-ARINC, ont notamment été utilisées.

Nous prévoyons d'étendre cette modélisation afin de couvrir un périmètre plus large de AADL, ainsi que des aspects temporels supplémentaires qui sont liés à la conception GALS et permettent des analyses temporelles plus sophistiquées. Des connexions à d'autres outils de simulation pour ce qui concerne l'analyse temporelle sont aussi prévues, comme par exemple SynDEx ou RT-Builder. Enfin, des extensions de notre modèle d'ordonnancement constituent aussi une perspective à ce travail.

8. Bibliographie

Anderson R., *Security Engineering : A Guide to Building Dependable Distributed Systems*, Wiley, Computer Laboratory, University of Cambridge, 2001.

ARINC651, « Design Guidance for Integrated Modular Avionics », *Airlines Electronic Engineering Committee, ARINC Report 651-1*, 1997.

ARINC653, « Avionics Application Software Standard Interface », *Airlines Electronic Engineering Committee, ARINC Specification 653*, 1997.

Aubry P., Le Guernic P., Machard S., « Synchronous Distribution of SIGNAL Programs », *Hawaii International Conference on System Sciences*, vol. 0, p. 656, 1996.

Benveniste A., Le Guernic P., Jacquemot C., « Synchronous programming with events and relations : the SIGNAL language and its semantics », *Science of Computer Programming*p. 16 :103-149, 1991.

Besnard L., Gautier T., Le Guernic P., « SIGNAL V4-Inria Version : Reference manual », 2011.

Daniel M. C., Globally-Asynchronous Locally-Synchronous Systems, PhD thesis, Stanford University, 1984.

Delange J., Pautet L., Plantec A., Kerboeuf M., Singhoff F., Kordon F., « Validate, Simulate and Implement ARINC653 Systems using the AADL », *ACM SIGAda Ada Letters. Volume 29. Number 3, Pages 31-44. ISSN :1094-3641*, November, 2009.

ESPRESSO, INRIA, « SME », 2011. `http://www.irisa.fr/espresso/Polychrony/`.

Gamatié A., Modélisation polychrone et évaluation de systèmes temps réel, PhD thesis, IFSIC/IRISA, 2004.

Gamatié A., Gautier T., « Modeling of avionics applications and performance evaluation techniques using the synchronous language Signal », *SLAP'03*, Elsevier, 2003.

Jahier E., Halbwachs N., Raymond P., Nicollin X., Lesens D., « Virtual Execution of AADL Models via a Translation into Synchronous Programs », *EMSOFT2007*, Salzburg Austria, p. 134 - 143, 2007. ASSERT.

Knight J. C., « Safety Critical Systems : Challenges and Directions », *ICSE 2002*, 2002.

Laplante P. A., *Real-Time Systems Design and Analysis : An Engineer's Handbook*, IEEE Press, Piscataway, NJ, USA, 1992.

Le Guernic P., Talpin J.-P., Le Lann J.-C., « Polychrony for System Design », *Journal for Circuits, Systems and Computers*, vol. 12, p. 261-304, 2002.

Lee S.-Y., Mallet F., de Simone R., « Dealing with AADL End-to-End Flow Latency with UML MARTE », *Engineering of Complex Computer Systems*p. 228-233, 2008.

Ma Y., Compositional modeling of globally asynchronous locally synchronous (GALS) architectures in a polychronous model of computation, PhD thesis, University of Rennes1, France, 2010.

Ma Y., Talpin J.-P., Gautier T., « Virtual prototyping AADL architectures in a polychronous model of computation », *MEMOCODE08*, 2008.

Ma Y., Talpin J.-P., Shukla S. K., Gautier T., « Distributed Simulation of AADL Specifications in a Polychronous Model of Computation », *ICESS'09*, 2009.

Marchand H., Bournai P., Le Borgne M., Le Guernic P., « Synthesis of Discrete-Event Controllers based on the Signal Environment », *in Discrete Event Dynamic System : Theory and Applications*, p. 325-346, 2000.

Pi L., Bodeveix J.-P., Filali M., « Modeling AADL Data Communication with BIP », *Ada-Europe'09*, Springer-Verlag, Berlin, Heidelberg, p. 192-206, 2009.

SAE AS5506, « Architecture Analysis and Design Language (AADL) », 2004.

SAE AS5506A, « Architecture Analysis and Design Language (AADL) (v2) », 2009.

Talpin J.-P., Le Guernic P., Shukla S. K., Gupta R., Doucet F., « Polychrony for Formal Refinement-Checking in a System-Level Design Methodology », 2003.

Talpin J.-P., Ouy J., Besnard L., Le Guernic P., « Compositional design of isochronous systems », *DATE '08*, ACM, New York, NY, USA, p. 928-933, 2008.

Yu H., Ma Y., Glouche Y., Talpin J.-P., Besnard L., Gautier T., Le Guernic P., Toom A., Laurent O., « System-level Co-simulation of Integrated Avionics Using Polychrony », *SAC'11*, Taiwan, March, 2011.